chapter

12

Variables and Operators

12.1 Introduction

In this chapter, we cover two basic concepts of high-level language programming, variables and operators. *Variables* hold the values upon which a program acts, and *operators* are the language mechanisms for manipulating these values. Variables and operators together allow the programmer to more easily express the work that a program is to carry out.

The following line of C code is a statement that involves both variables and operators. In this statement, the addition operator + is used to add 3 to the original value of the variable `score`. This new value is then assigned using the assignment operator = back to `score`. If `score` was equal to 7 before this statement was executed, it would equal 10 afterwards.

```
score = score + 3;
```

In the first part of this chapter, we'll take a closer look at variables in the C programming language. Variables in C are straightforward: the three most basic flavors are integers, characters, and floating point numbers. After variables, we'll cover C's rich set of operators, providing plenty of examples to help illustrate their operations. One unique feature of our approach is that we can connect both of these high-level concepts back to solid low-level material, and in the third part of the chapter we'll do just that by discussing the compiler's point of view when it tries to deal with variables and operators in generating machine code. We close this chapter with some problem solving and some miscellaneous concepts involving variables and operators in C.

12.2 Variables

A value is any data item upon which a program performs an operation. Examples of values include the iteration counter for a loop, an input value entered by a user, or the partial sum of a series of numbers that are being added together. Programmers spend a lot of effort keeping track of these values.

Because values are such an important programming concept, high-level languages try to make the process of managing them easier on the programmer. High-level languages allow the programmer to refer to values *symbolically*, by a name rather than a memory location. And whenever we want to operate on the value, the language will automatically generate the proper sequence of data movement operations. The programmer can then focus on writing the program and need not worry about where in memory to store a value or about juggling the value between memory and the registers. In high-level languages, these symbolically named values are called *variables*.

In order to properly track the variables in a program, the high-level language translator (the C compiler, for instance) needs to know several characteristics about each variable. It needs to know, obviously, the symbolic name of the variable. It needs to know what type of information the variable will contain. It needs to know where in the program the variable will be accessible. In most languages, C included, this information is provided by the variable's *declaration*.

Let's look at an example. The following declares a variable called `echo` that will contain an integer value.

```
int echo;
```

Based on this declaration, the compiler reserves an integer's worth of memory for `echo` (sometimes, the compiler can optimize the program such that `echo` is stored in a register and therefore does not require a memory location, but that is a subject for a later course). Whenever `echo` is referred to in the subsequent C code, the compiler generates the appropriate machine code to access it.

12.2.1 Three Basic Data Types: `int`, `char`, `double`

By now, you should be very familiar with the following concept: the meaning of a particular bit pattern depends on the data type imposed on the pattern. For example, the binary pattern `0110 0110` might represent the lowercase `f` or it might represent the decimal number 102, depending on whether we treat the pattern as an ASCII data type or as a 2's complement integer data type. A variable's declaration informs the compiler about the variable's type. The compiler uses a variable's type information to allocate a proper amount of storage for the variable. Also, type indicates how operations on the variable are to be performed at the machine level. For instance, performing an addition on two integer variables can be done on the LC-3 with one ADD instruction. If the two variables were of floating point type, the LC-3 compiler would generate a sequence of instructions to perform the addition because no single LC-3 instruction performs a floating point addition.

C supports three basic data types: integers, characters, and floating point numbers. Variables of these types can be created with the type specifiers `int`, `char`, and `double` (which is short for *double*-precision floating point).

int

The `int` type specifier declares a signed integer variable. The internal representation and range of values of an `int` depends on the ISA of the computer and the specifics of the compiler being used. In the LC-3, for example, an `int` is a 16-bit 2's complement integer that can represent numbers between −32,768 and +32,767. On an x86-based computer, an `int` is likely to be a 32-bit 2's complement number that can represent numbers between −2,147,483,648 and +2,147,483,647. In most cases, an `int` is a 2's complement integer in the word length of the underlying ISA.

The following line of code declares an integer variable called `numberOfSeconds`. When the compiler sees this declaration, the compiler sets aside enough storage for this variable (in the case of the LC-3, one memory location).

```
int numberOfSeconds;
```

It should be no surprise that variables of integer type are frequently used in programs. They often conveniently represent the real-world data we want our programs to process. If we wanted to represent time, say for example in seconds, an integer variable would be perfect. In an application that tracks whale migration, we can use an integer to represent the sizes of pods of gray whales seen off the California coast. Integers are also useful for program control. An integer can be useful as the iteration counter for a counter-controlled loop.

char

The `char` type specifier declares a variable whose data value represents a character. Following are two examples. The first declaration creates a variable named `lock`. The second one declares `key`. The second declaration is slightly different; it also contains an *initializer*. In C, any variables can be set to an initial value directly in its declaration. In this example, the variable `key` will have the initial value of the ASCII code for uppercase *Q*. Also notice that the uppercase *Q* is surrounded by single quotes, `' '`. In C, characters that are to be interpreted as ASCII *literals* are surrounded by single quotes. What about `lock`? What initial value will it have? We'll address this issue shortly.

```
char lock;
char key = 'Q';
```

Although eight bits are sufficient to hold an ASCII character, for purposes of making the examples in this textbook less cluttered, all `char` variables will occupy 16 bits. That is, `char`s, like `int`s, will each occupy one memory location.

double

The type specifier `double` allows us to declare variables of the floating point type that we examined in Section 2.7.2. Floating point numbers allow us to

conveniently deal with numbers that have fractional components or numbers that are very large or very small. Recall from our previous discussion in Section 2.7.2 that at the lowest level, a floating point number is a bit pattern that has three parts: a sign, a fraction, and an exponent.

Here are three examples of variables of type `double`:

```
double costPerLiter;
double electronsPerSecond;
double averageTemp;
```

As with `int`s and `char`s, we can also optionally initialize a floating point number along with its declaration. Before we can completely describe how to initialize floating point variables, we must first discuss how to represent floating point *literals* in C. Floating point literals are represented containing either a decimal point or an exponent, or both, as demonstrated in the example code that follows. The exponent is signified by the character *e* or *E* and can be positive or negative. It represents the power of 10 by which the fractional part (the part that precedes the *e* or *E*) is multiplied. Note that the exponent must be an integer value. For more information on floating point literals, see Appendix D.2.4.

```
double twoPointOne = 2.1;          /* This is 2.1   */
double twoHundredTen = 2.1E2;      /* This is 210.0 */
double twoHundred = 2E2;           /* This is 200.0 */
double twoTenths = 2E-1;           /* This is 0.2   */
double minusTwoTenths = -2E-1;     /* This is -0.2  */
```

Another floating point type specifier in C is called `float`. It declares a single-precision floating point variable; `double` creates one that is double-precision. Recall from our previous discussion on floating point numbers in Chapter 2 that the precision of a floating point number depends on the number of bits of the representation allocated to the fraction. In C, depending on the compiler and the ISA, a `double` may have more bits allocated for the fraction than a `float`, but never fewer. The size of the `double` is dependent upon the ISA and the compiler. Usually, a `double` is 64 bits long and a `float` is 32 bits in compliance with the IEEE 754 floating point standard.

12.2.2 Choosing Identifiers

Most high-level languages have flexible rules for the variable names (more generally known as *identifiers*) that can be chosen within a program. C allows you to create identifiers composed of letters of the alphabet, digits, and the underscore character, _. Only letters and the underscore character, however, can be used to begin an identifier. An identifier can be of any length, but only the first 31 characters are used by the C compiler to differentiate variables—only the first 31 characters matter to the compiler. Also, the use of upper- and lowercase has significance: C will treat `Capital` and `capital` as different indentifiers.

Here are several tips on standard C naming conventions: Variables beginning with an underscore (e.g., `_index_`) conventionally are used only in special library code. Variables are almost never declared in all uppercase letters. The convention of all uppercase is used solely for symbolic values created using the preprocessor directive `#define`. See Section 11.5.3 for examples of symbolic constants. Programmers like to visually partition variables that consist of multiple words. In this book, we use uppercase (e.g., `wordsPerSecond`). Other programmers prefer underscores (e.g., `words_per_second`).

Giving variables meaningful names is important for writing good code. Variable names should be chosen to reflect a characteristic of the value they represent, allowing the programmer to more easily recall what the value is used for. For example, a value used to count the number of words the person at the keyboard types per second might be named `wordsPerSecond`.

There are certain *keywords* in C that have special meaning and are therefore restricted from being used as identifiers. A list of C keywords can be found in Appendix D.2.6. One keyword we have encountered already is `int`, and therefore we cannot use `int` as a variable name. Having a variable named `int` would not only be confusing to someone trying to read through the code but might also confuse the compiler trying to translate it. The compiler may not be able to determine whether a particular `int` refers to the variable or to the type specifier.

12.2.3 Scope: Local versus Global

As we mentioned, a variable's declaration assists the compiler in managing the storage of that variable. In C, a variable's declaration conveys three pieces of information to the compiler: the variable's *identifier*, its *type*, and its *scope*. The first two of these, identifier and type, the C compiler gets explicitly from the variable's declaration. The third piece, scope, the compiler infers from the position of the declaration within the code. The scope of a variable is the region of the program in which the variable is "alive" and accessible.

The good news is that in C, there are only two basic types of scope for a variable. Either the variable is *global* to the entire program,[1] or it is *local*, or private, to a particular block of code.

Local Variables

In C, all variables must be declared before they can be used. In fact, some variables must be declared at the beginning of the *block* in which they appear—these are called local variables. In C, a *block* is any subsection of a program beginning with the open brace character, `{` and ending with the closing brace character, `}`. All local variables must be declared immediately following the block's open brace.

The following code is a simple C program that gets a number from the keyboard and redisplays it on the screen. The integer variable `echo` is declared within

[1] This is a slight simplification because C allows globals to be optionally declared to be global only to a particular source file and not the entire program, but this caveat is not relevant for our discussion here.

the block that contains the code for function `main`. It is only visible to the function `main`. If the program contained any other functions besides `main`, the variable would not be accessible from those other functions. Typically, most local variables are declared at the beginning of the function in which they are used, as for example `echo` in the code.

```
#include <stdio.h>

int main()
{
    int echo;

    scanf("%d", &echo);
    printf("%d\n", echo);
}
```

It is possible, and sometimes useful, to declare two different variables with the same name within different blocks of the same function. For instance, it might be convenient to use the name `count` for the counter variable for several different loops within the same program. C allows this, as long as the different variables sharing the same name are declared in seperate blocks. Figure 12.1, which we discuss in the next section, provides an example of this.

Global Variables

In contrast to local variables, which can only be accessed within the block in which they are declared, global variables can be accessed throughout the program. They retain their storage and values throughout the duration of the program.

```
#include <stdio.h>

int globalVar = 2;                    /* This variable is global */

int main()
{
  int localVar = 3;    /* This variable is local to main */

  printf("Global %d Local %d\n", globalVar, localVar);

  {
    int localVar = 4;                 /* Local to this sub-block */

    printf("Global %d Local %d\n", globalVar, localVar);
  }

  printf("Global %d Local %d\n", globalVar, localVar);
}
```

Figure 12.1 A C program that demonstrates nested scope

The following code contains both a global variable and a variable local to the function `main`:

```
#include <stdio.h>

int globalVar = 2;                    /* This variable is global */

int main()
{
  int localVar = 3;    /* This variable is local to main */

  printf("Global %d Local %d\n", globalVar, localVar);
}
```

Globals can be extremely helpful in certain programming situations, but novice programmers are often instructed to adopt a programming style that uses locals over globals. Because global variables are public and can be modified from anywhere within the code, the heavy use of globals can make your code more vulnerable to bugs and more difficult to reuse and modify. In almost all C code examples in this textbook, we use only local variables.

Let's look at a slightly more complex example. The C program in Figure 12.1 is similar to the previous program except we have added a sub-block within `main`. Within this sub-block, we have declared a new variable `localVar`. It has the same name as the local variable declared at the beginning of `main`. Execute this program and you will notice that when the sub-block is executing the prior version of `localVar` is not visible; that is, the new declaration of a variable of the same name supersedes the previous one. Once the sub-block is done executing, the previous version of `localVar` becomes visible again. This is an example of what is called *nested scope*.

Initialization of Variables

Now that we have discussed global and local variables, let's answer the question we asked earlier: What initial value will a variable have if it has no initializer? In C, by default, local variables start with an unknown value. That is, the storage location a local variable is assigned is not cleared and thus contains whatever last value was stored there. More generally, in C, local variables are uninitialized (in particular, all variables of the *automatic storage class*). Global variables (and all other static *storage class variables*) are, in contrast, initialized to 0 when the program starts execution.

12.2.4 More Examples

Let's examine a couple more examples of variable declarations in C. The following examples demonstrate declarations of the three basic types discussed in this chapter. Some declarations have no initializers; some do. Notice how floating point and character literals are expressed in C.

```
double width;
double pType = 9.44;
double mass = 6.34E2;
double verySmallAmount = 9.1094E-31;
double veryLargeAmount = 7.334553E102;
int average = 12;
int windChillIndex = -21;
int unknownValue;
int mysteryAmount;
char car    = 'A';
char number = '4';
```

In C, it is also possible to have literals that are hexadecimal values. A literal that has the prefix `0x` will be treated as a hexadecimal number. In the following examples, all three integer variables are initialized using hexadecimal literals.

```
int programCounter = 0x3000;
int sevenBits = 0xA1234;
int valueD = 0xD;
```

Questions: What happens if we perform a `printf("%d\n", valueD);` after the declarations? What bit pattern would you expect to find in the memory location associated with `valueD`?

12.3 Operators

Having covered the basics of variables in C, we are now ready to investigate operators. C, like many other high-level languages, supports a rich set of operators that allow the programmer to manipulate variables. Some operators perform arithmetic, some perform logic functions, and others perform comparisons between values. These operators allow the programmer to express a computation in a more natural, convenient, and compact way than by expressing it as a sequence of assembly language instructions.

Given some C code, the compiler's job is to take the code and convert it into machine code that the underlying hardware can execute. In the case of a C program being compiled for the LC-3, the compiler must translate whatever operations the program might contain into the instructions of the LC-3 instruction set—clearly not an easy task given that the LC-3 has very few operate instructions.

To help illustrate this point, we examine the code generated by a simple C statement in which two integers are multiplied together. In the following code segment, `x`, `y`, and `z` are integer variables where `x` and `y` are multiplied and the result *assigned* to `z`.

```
z = x * y;
```

Since there is no single LC-3 instruction to multiply two values, our LC-3 compiler must generate a sequence of code that accomplishes the multiplication of

```
        AND   R0, R0, #0    ;   R0 <= 0

        LDR   R1, R5, #0    ;   load value of x
        LDR   R2, R5, #-1   ;   load value of y
        BRz   DONE          ;   if y is zero, we're done
        BRp   LOOP          ;   if y is positive, start mult

                            ;   y is negative
        NOT   R1, R1        ;
        ADD   R1, R1, #1    ;   R1 <= -x

        NOT   R2, R2
        ADD   R2, R2, #1    ;   R2 <= -y (-y is positive)

LOOP    ADD   R0, R0, R1    ;   Multiply loop
        ADD   R2, R2, #-1   ;   The result is in R2
        BRp   LOOP

DONE:   STR   R0, R5, #-2   ;   z = x * y;
```

Figure 12.2 The LC-3 code for C multiplication

two (possibly negative) integers. One possible manner in which this can be accomplished is by repeatedly adding the value of `x` to itself a total of `y` times. This code is similar to the code in the calculator example in Chapter 10. Figure 12.2 lists the resulting LC-3 code generated by the LC-3 compiler. Assume that register 5 (R5) contains the memory address where variable `x` is allocated. Immediately prior to that location is where variable `y` is allocated (i.e., R5 − 1), and immediately prior to that is where variable `z` resides. While this method of allocating variables in memory might seem a little strange at first, we will explain this later in Section 12.5.2.

12.3.1 Expressions and Statements

Before proceeding with our coverage of operators, we'll diverge a little into C syntax to help clarify some syntactic notations used within C programs. We can combine variables and literal values with operators, such as the multiply operator from the previous example, to form a C *expression*. In the previous example, `x * y` is an expression.

Expressions can be grouped together to form a *statement*. For example, `z = x * y;` is a statement. Statements in C are like complete sentences in English. Just as a sentence captures a complete thought or action, a C statement expresses a complete unit of work to be carried out by the computer. All statements in C end with a semicolon character, `;` (or as we'll see in the next paragraph, a closing brace, `}`). The semicolon terminates the end of a statement in much the same way a punctuation mark terminates a sentence in English. An interesting (or perhaps odd) feature of C is that it is possible to create statements that do not express any computation but are syntactically considered statements. The null statement is simply a semicolon and it accomplishes nothing.

One or more simple statements can be grouped together to form a compound statement, or *block*, by enclosing the simple statements within braces, { }. Syntactically, compound statements are equivalent to simple statements. We will see many real uses of compound statements in the next chapter.

The following examples show some simple, compound, and null statements.

```
z = x * y;     /* This statement accomplishes some work */

{              /* This is a compound statement          */
  a = b + c;
  i = p * r * t;
}

k = k + 1;     /* This is another simple statement      */
;              /* Null statement -- no work done here   */
```

12.3.2 The Assignment Operator

We've already seen examples of C's assignment operator. Its symbol is the equal sign, =. The operator works by first evaluating the right-hand side of the assignment, and then assigning the value of the right-hand side to the *object* on the left-hand side. For example, in the C statement

```
a = b + c;
```

the value of variable `a` will be set equal to the value of the expression `b + c`.

Notice that even though the arithmetic symbol for equality is the same as the C symbol for assignment, they have different meanings. In mathematics, by using the equal sign, =, one is making the assertion that the right-hand and left-hand expressions are equivalent. In C, using the = operator causes the compiler to generate code that will make the left-hand side change its value to equal the value of the right-hand side. In other words, the left-hand side is *assigned* the value of the right-hand side.

Let's examine what happens when the LC-3 C compiler generates code for a statement containing the assignment operator. The C following statement represents the increment by 4 of the integer variable `x`.

```
x = x + 4;
```

The LC-3 code for this statement is straightforward. Here, R5 contains the address of variable `x`.

```
LDR  R0, R5, #0   ;   Get the value of x
ADD  R0, R0, #4   ;   calculate x + 4
STR  R0, R5, #0   ;   x = x + 4;
```

In C, all expressions evaluate to a value of a particular type. From the previous example, the expression `x + 4` evaluates to an integral value because we

are adding an integer 4 to another integer (the variable x). This integer result is then assigned to an integer variable. But what would happen if we constructed an expression of mixed type, for example `x + 4.3`? The general rule in C is that the mixed expressions like the one shown will be *converted* from integer to floating point. If an expression contains both integer and character types, it will be promoted to integer type. In general, in C shorter types are converted to longer types. What if we tried to assign an expression of one type to a variable of another, for example `x = x + 4.3`? In C, the type of a variable remains immutable (meaning it cannot be changed), so the expression is converted to the type of the variable. In this case, the floating point expression `x + 4.3` is converted to integer. In C, floating point values are rounded into integers by dropping the fractional part. For example, 4.3 will be rounded to 4 when converting from a floating point into an integer; 5.9 will be rounded to 5.

12.3.3 Arithmetic Operators

The arithmetic operators are easy to understand. Many of the operations and corresponding symbols are ones to which we are accustomed, having used them since learning arithmetic in grade school. For instance, `+` performs addition, `-` subtraction, `*` performs multiplication (which is different from the symbol we are accustomed to for multiplication in order to avoid confusion with the letter x), and `/` performs division. Just as when doing arithmetic by hand, there is an order in which expressions are evaluated. Multiplication and division are evaluated first, followed by addition and subtraction. The order in which operators are evaluated is called *precedence*, and we discuss it in more detail in the next section. Following are several C statements formed using the arithmetic operators:

```
distance = rate * time;
netIncome = income - taxesPaid;
fuelEconomy = milesTraveled / fuelConsumed;
area = 3.14159 * radius * radius;
y = a*x*x + b*x + c;
```

C has another arithmetic operator that might not be as familiar to you as `+`, `-`, `*`, and `/`. It is the *modulus* operator, `%` (also known as the integer remainder operator). To illustrate its operation, consider what happens when we divide two integer values. When performing an integer divide in C, the fractional part is dropped and the integral part is the result. The expression `11 / 4` evaluates to 2. The modulus operator `%` can be used to calculate the integer remainder. For example, `11 % 4` evaluates to 3. Said another way, `(11 / 4) * 4 + (11 % 4)` is equal to 11. In the following example, all variables are integers.

```
quotient  = x / y;  /* if x = 7 and y = 2, quotient = 3  */
remainder = x % y;  /* if x = 7 and y = 2, remainder = 1 */
```

Table 12.1 lists all the arithmetic operations and their symbols. Multiplication, division, and modulus have higher precedence than addition and subtraction.

Table 12.1 Arithmetic Operators in C

Operator symbol	Operation	Example usage
*	multiplication	x * y
/	division	x / y
%	modulus	x % y
+	addition	x + y
-	subtraction	x - y

12.3.4 Order of Evaluation

Before proceeding onwards to the next set of C operators, we diverge momentarily to answer an important question: What value is stored in x as a result of the following statement?

```
x = 2 + 3 * 4;
```

Precedence

Just as when doing arithmetic by hand, there is an order to which expressions are evaluated. And this order is called operator *precedence*. For instance, when doing arithmetic, multiplication and division have higher precedence than addition and subtraction. For the arithmetic operators, the C precedence rules are the same as we were taught in grade-school arithmetic. In the preceding statement, x is assigned the value 14 because the multiplication operator has higher precedence than addition. That is, the expression evaluates as if it were $2 + (3 * 4)$.

Associativity

But what about operators of equal precedence? What does the following statement evaluate to?

```
x = 2 + 3 - 4 + 5;
```

Depending on which operator we evaluate first, the value of the expression $2 + 3 - 4 + 5$ could equal 6 or it could equal -4. Since the precedence of both operators is the same (that is, addition has the same precedence as subtraction in C), we clearly need a rule on how such expressions should be evaluated in C. For operations of equal precedence, their *associativity* determines the order in which they are evaluated. In the case of addition and subtraction, both associate from left to right. Therefore $2 + 3 - 4 + 5$ evaluates as if it were $((2 + 3) - 4) + 5$.

The complete set of precedence and associativity rules for all operators in C is provided in Table 12.5 at the end of this chapter and also in Table D.4. We suggest that you do not try to memorize this table (unless you enjoy quoting C trivia to your friends). Instead, it is important to realize that the precedence rules exist and to roughly comprehend the logic behind them. You can always refer to the table whenever you need to know the relationship between particular operators. There is a safeguard, however: parentheses.

Parentheses

Parentheses override the evaluation rules by specifying explicitly which operations are to be performed ahead of others. As in arithmetic, evaluation always begins at the innermost set of parentheses. We can surround a subexpression with parentheses if we want that subexpression to be evaluated first. So in the following example, say the variables `a`, `b`, `c`, and `d` are all equal to 4. The statement

```
x = a * b + c * d / 2;
```

could be written equivalently as

```
x = (a * b) + ((c * d) / 4);
```

For both statements, `x` is set to the value of 20. Here the program will always evaluate the innermost subexpression first and move outward before falling back on the precedence rules.

What value would the following expression evaluate to if `a`, `b`, `c`, and `d` equal 4?

```
x = a * (b + c) * d / 4;
```

Parentheses can help make code more readable, too. Most people reading your code are unlikely to have memorized C's precedence rules. For this reason, for long or complex expressions, it is often stylistically preferable to use parentheses, even if the code works fine without them.

12.3.5 Bitwise Operators

We now return to our discussion of C operators. C has a set of operators called *bitwise* operators that manipulate bits of a value. That is, they perform a logical operation such as AND, OR, NOT, XOR across the individual bits of a value. For example, the C bitwise operator `&` performs an operation similar to the LC-3 AND instruction. That is, the `&` operator performs an AND operation bit by bit across the two input operands. The C operator `|` performs a bitwise OR. The operator `~` performs a bitwise NOT and takes only one operand (i.e., it is a unary operator). The operator `^` performs a bitwise XOR. Examples of expressions using these operators on 16-bit values follow.

```
0x1234 | 0x5678   /* equals 0x567C */
0x1234 & 0x5678   /* equals 0x1230 */
0x1234 ^ 0x5678   /* equals 0x444C */
~0x1234           /* equals 0xEDCB */
1234 & 5678       /* equals 1026   */
```

C's set of bitwise operators includes two shift operators: `<<`, which performs a left shift, and `>>`, which performs a right shift. Both are binary operators, meaning they require two operands. The first operand is the value to be shifted and the second operand indicates the number of bit positions to shift by. On a left shift, the vacated bit positions of the value are filled with zeros; on a right shift, the value is sign-extended. The result is the value of the expression; neither of the

Table 12.2 Bitwise Operators in C

Operator symbol	Operation	Example usage
~	bitwise NOT	~x
«	left shift	x « y
»	right shift	x » y
&	bitwise AND	x & y
^	bitwise XOR	x ^ y
|	bitwise OR	x | y

two original operand values are modified. The following expressions provide examples of these two operators operating on 16-bit integers.

```
0x1234 << 3      /* equals 0x91A0  */
0x1234 >> 2      /* equals 0x048D  */
1234 << 3        /* equals 9872    */
1234 >> 2        /* equals 308     */
0x1234 << 5      /* equals 0x4680  (result is 16 bits)    */
0xFEDC >> 3      /* equals 0xFFDB  (from sign-extension) */
```

Here we show several C statements formed using the bitwise operators. For all of C's bitwise operators, neither operand can be a floating point value. For these statements, `f`, `g`, and `h` are integers.

```
h = f & g;      /* if f = 7, g = 8, h will equal 0      */
h = f | g;      /* if f = 7, g = 8, h will equal 15     */
h = f << 1;     /* if f = 7, g = 8, h will equal 14     */
h = g << f;     /* if f = 7, g = 8, h will equal 1024   */
h = ~f | ~g;    /* if f = 7, g = 8, h will equal -1     */
                /* because h is a signed integer        */
```

Question: Say that on a particular machine, the integer `x` occupies 16 bits and has the value 1. What happens after the statement `x = x << 16;` is executed? Conceptually, we are shifting `x` by its data width, replacing all bits with 0. You might expect the value of `x` to be 0. To remain generic, C formally defines the result of shifting a value by its width (or more than its data width) as implementation-dependent. This means that the result might be 0 or it might not, depending on the system on which the code is executed.

Table 12.2 lists all the bitwise operations and their symbols. The operators are listed in order of precedence, the NOT operator having highest precedence, and the left and right shift operators having equal precedence, followed by AND, then XOR, then OR. They all associate from left to right. See Table 12.5 for a complete listing of operator precedence.

12.3.6 Relational Operators

C has several operators to test the relationship between two values. As we will see in the next chapter, these operators are often used in C to generate conditional

Table 12.3 Relational Operators in C

Operator symbol	Operation	Example usage
>	greater than	x > y
>=	greater than or equal	x >= y
<	less than	x < y
<=	less than or equal	x <= y
==	equal	x == y
!=	not equal	x != y

constructs (similar to the conditional constructs we discussed in Section 6.1.2 when we discussed systematic decomposition).

The equality operator, `==`, is one of C's relational operators. This operator tests if two values are equal. If they are equal, the expression evaluates to a 1, and if they are not, the expression evaluates to 0. The following shows two examples:

```
q = (312 == 83);    /* q will equal 0 */
z = (x == y);       /* z will equal 1 if x equals y */
```

In the second example, the right-hand side of the assignment operator `=` is the expression `x == y`, which evaluates to a 1 or a 0, depending on whether `x` and `y` are equal. (Note: The parentheses are not required because the `==` operator has higher precedence than the `=` operator. We added them to help make the example clearer).

Opposite the equality operator, the inequality operator, `!=`, evaluates to a 1 if the operands are not equal. Other relational operators test for greater than, less than, and so on, as described in the following examples. For these examples, the variables `f`, `g`, and `h` are integers. The variable `f` has the value 7, and `g` is 8.

```
h = f == g;     /* Equal To operator.      h will equal 0  */
h = f > g;      /* Greater Than operator.  h will equal 0  */
h = f != g;     /* Not Equal To operator.  h will equal 1  */
h = f <= g;     /* Less Than Or Equal To.  h will equal 1  */
```

The next example is a preview of coming attractions. The C relational operators are very useful for performing tests on variables in order to change the flow of the program. In the next chapter, we describe the C `if` statement in more detail. However, the concept of an `if` construct is not a new one—we have been dealing with this particular decision construct ever since learning how to program the LC-3 in Chapter 6. Here, a message is printed only if the variable `tankLevel` is equal to zero.

```
if (tankLevel == 0)
   printf("Warning: Tank Empty!!\n");
```

Table 12.3 lists all the relational operators and provides a simple example of each. The first four operators have higher precedence than the last two. Both sets associate from left to right.

12.3.7 Logical Operators

C's logical operators appear at first glance to be exactly like some of the bitwise operators, and many novice programmers sometimes confuse the two. Before we explain their operation, we need to mention C's concept of logically true and logically false values. C adopts the notion that a nonzero value (i.e., a value other than zero) is logically true. A value of zero is logically false. It is an important concept to remember, and we will see it surface many times as we go through the various components of the C language.

C supports three logical operators: `&&`, `||`, and `!`. The `&&` operator performs a logical AND of its two operands; it evaluates to an integer value of 1 (which is logically true) if both of its operands are logically true, or nonzero. It evaluates to 0 otherwise. For example, 3 `&&` 4 evaluates to a 1, whereas 3 `&&` 0 evaluates to 0. The `||` operator is C's logical OR operator. The expression `x || y` evaluates to a 1 if either `x` OR `y` are nonzero. For example, 3 `||` 4 evaluates to a 1. Also, 3 `||` 0 evaluates to 1. The negation operator `!` evalutes to the other logical state of its operand. So `!x` is 1 only if `x` equals 0. It is 0 otherwise.

What are the logical operators useful for? One use is for constructing logical conditions within a program. For example, we can determine if a variable is within a particular range of values using a combination of relational and logical operators. To check if `x` is between 10 and 20, inclusive, we can use the following expression:

```
(10 <= x) && (x <= 20)
```

Or to test if a character `c` is a letter of the alphabet:

```
(('a' <= c) && (c <= 'z')) || (('A' <= c) && (c <= 'Z'))
```

Here are some examples of the logical operators, with several previous examples of bitwise operators included to highlight the difference. As in the previous examples, the variables `f`, `g`, and `h` are integers. The variable `f` has the value 7, and `g` is 8.

```
h = f & g;      /* bitwise operator: h will equal 0  */
h = f && g;     /* logical operator: h will equal 1  */
h = f | g;      /* bitwise operator: h will equal 15 */
h = f || g;     /* logical operator: h will equal 1  */
h = ~f | ~g;    /* bitwise operator: h will equal -1 */
h = !f && !g;   /* logical operator: h will equal 0  */
h = 29 || -52;  /* logical operator: h will equal 1  */
```

Table 12.4 lists logical operators in C and their symbols. The logical NOT operator has highest precedence, then logical AND, then logical OR. See Table 12.5 for a complete listing of operator precedence.

12.3.8 Increment/Decrement Operators

Because incrementing and decrementing variables is such a commonly performed operation, the designers of the C programming language decided to include special

Table 12.4 Logical Operators in C

Operator symbol	Operation	Example usage
!	logical NOT	`!x`
&&	logical AND	`x && y`
||	logical OR	`x || y`

Table 12.5 Operator Precedence, from Highest to Lowest. Descriptions of Some Operators are Provided in Parentheses

Precedence	Associativity	Operators
1 (highest)	l to r	`()` (function call) `[ ]` (array index) `.` `->`
2	r to l	`++` `--` (postfix versions)
3	r to l	`++` `--` (prefix versions)
4	r to l	`*` (indirection) `&` (address of) `+` (unary) `-` (unary) `~` `!` `sizeof` `(type)` (type cast)
5	r to l	`*` (multiplication) `/` `%`
6	l to r	`+` (addition) `-` (subtraction)
7	l to r	`<<` `>>`
8	l to r	`<` `>` `<=` `>=`
9	l to r	`==` `!=`
10	l to r	`&`
11	l to r	`^`
12	l to r	`|`
13	l to r	`&&`
14	l to r	`||`
15	l to r	`?:` (conditional expression)
16	l to r	`=` `+=` `-=` `*=` etc.
17 (lowest)	r to l	

operators to perform them. The `++` operator *increments* a variable to the next higher value. The `--` operator *decrements* it. For example, the expression `x++` increments the value of integer variable `x` by 1. The expression `x--` decrements the value of `x` by 1. Keep in mind that these operators modify the value of the variable itself. That is, `x++` is similar to the operation `x = x + 1`.

The `++` and `--` operators can be used on either side of a variable. The expression `++x` operates in a slightly different order than `x++`. If `x++` is part of a larger expression, then the value of `x++` is the value of `x` prior to the increment, whereas the value of `++x` is the incremented value of `x`. If the operator `++` appears before the variable, then it is used in *prefix* form. If it appears after the variable, it is in *postfix* form. The prefix forms are often referred to as *preincrement* and *predecrement*, whereas the postfix are *postincrement* and *postdecrement*.

Let's examine a couple of examples:

```
x = 4;
y = x++;
```

Here, the integer variable `x` is incremented. However, the original value of `x` is assigned to the variable `y` (i.e., the value of `x++` evaluates to the original

value of x). After this code executes, the variable y will have the value 4, and x will be 5.

Similarly, the following code increments x.

```
x = 4;
y = ++x;
```

However with this code, the expression ++x evaluates to the value after the increment. In this case, the value of both y and x will be 5.

This subtle distinction between the postfix and prefix forms is not too important to understand for now. For the few examples in this book that use these operators, the prefix and postfix forms of these operators can be used interchangeably. You can find a precise description of this difference in Appendix D.5.6.

12.3.9 Expressions with Multiple Operators

Thus far we've only seen examples of expressions with one or two operators. Real and useful expressions sometimes have more. We can combine various operators and operands to form complex expressions. The following example demonstrates a peculiar blend of operators forming a complex expression.

```
y = x & z + 3 || 9 - w % 6;
```

In order to figure out what this statement evaluates to, we need to examine the order of evaluation of operators. Table 12.5 lists all the C operators (including some that we have not yet covered but will cover later in this textbook) and their order of evaluation. According to precedence rules, this statement is equivalent to the following:

```
y = (x & (z + 3)) || (9 - (w % 6));
```

Another more useful expression that consists of multiple operators is given in the example that follows. In this example, if the value of the variable age is between 18 and 25, the expression evaluates to 1. Otherwise it is 0. Notice that even though the parentheses are not required to make the expression evaluate as we described, they do help make the code easier to read.

```
(18 <= age) && (age <= 25)
```

12.4 Problem Solving Using Operators

At this point, we have covered enough C operators to attempt a simple problem-solving exercise. For this problem, we will create a program that performs a simple network calculation: It calculates the amount of time required to transfer some number of bytes across a network with a particular transfer rate (provided in bytes per second). The twist to this problem is that transfer time is to be displayed as hours, minutes, and seconds.

We approach this problem by applying the decomposition techniques described in Chapter 6. That is, we will start with a very rough description of our program and continually refine it using the sequential, decision, and iteration constructs (see Chapter 6 if you need a refresher) until we arrive at something from which we can easily write C code. This technique is called *top-down*

decomposition because we start with a rough description of the algorithm and refine it by breaking larger steps into smaller ones, eventually arriving at something that resembles a program. Many experienced programmers rely on their understanding of the lower levels of the system to help make good decisions on how to decompose a problem. That is, in order to reduce a problem into a program, good programmers rely on their understanding of the basic primitives of systems they are programming on. In our case (at this point), these basic primitives are variables of the three C types and the operations we can perform on them.

In the subsequent chapters, we will go through several problem-solving examples to illustrate this top-down process. In doing so, we hope to provide you with a sense of the mental process a programmer might use to solve such problems.

The very first step (step 0) we need to consider for all problems from now on is how we represent the data items that the program will need to manipulate. At this point, we get to select from the three basic C types: integer, character, and floating point. For this problem, we can represent our internal calculations with either floating point values or integers. Since we are ultimately interested in displaying the result as hours, minutes, and seconds, any fractional components of

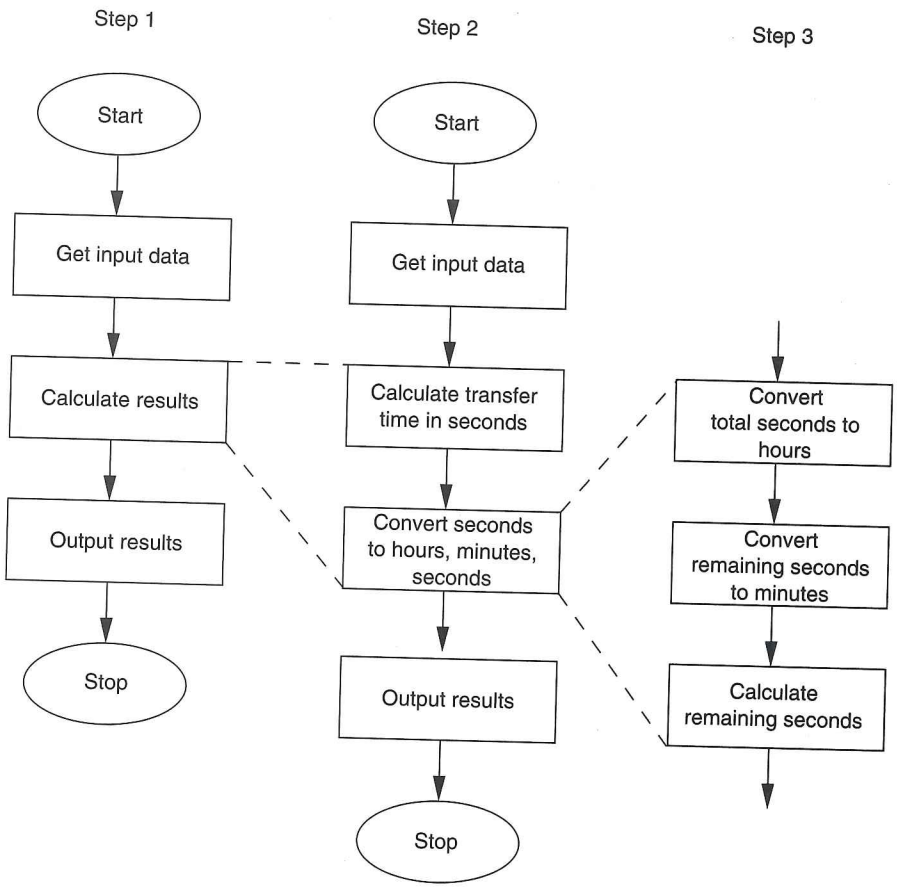


Figure 12.3 Stepwise refinement of a simple network transfer time problem

time are unnecessary. For example, displaying the total transfer time as 10.1 hours, 12.7 minutes, 9.3 seconds does not make sense. Rather, 10 hours, 18 minutes, 51 seconds is the preferred output. Because of this, the better choice of data type for the time calculation is integer (yes, there are rounding issues, but say we can ignore them for this calculation).

Having chosen our data representations, we can now apply stepwise refinement to decompose the problem. Figure 12.3 shows our decomposition of this particular programming problem. Step 1 in the figure shows the initial formulation of the problem. It involves three phases: get input, calculate results, output results. In the first phase, we will query the user about the amount of data to be transfered (in bytes) and the transfer rate of the network (in bytes per second). In the second phase, we will perform all necessary calculations, which we will then output in the third phase.

Step 1 is not detailed enough to translate directly into C code, and therefore we perform another refinement of it in step 2. Here we realize that the calculation phase can be further refined into a subphase that first calculates total time in seconds—which is an easy calculation given the input data—and a subphase to convert total time in seconds into hours, minutes, and seconds.

Step 2 is still not complete enough for mapping into C; we perform another refinement of it in step 3. Most phases of step 2 are fairly simple enough to convert into C, except for the conversion of seconds into hours, minutes, and seconds. In step 3, we refine this phase into three subphases. First we will calculate total hours based on the total number of seconds. Second, we will use the remaining seconds to calculate minutes. Finally, we determine the remaining number of seconds after the minutes have been calculated.

Based on the total breakdown of the problem after three steps of refinement presented in Figure 12.3, it should be fairly straightforward to map out the C code. The complete C program for this problem is presented in Figure 12.4.

12.5 Tying It All Together

We've now covered all the basic C types and operators that we plan to use throughout this textbook. Having completed this first exposure, we are now ready to examine these concepts from the compiler's viewpoint. That is, how does a compiler translate code containing variables and operators into machine code. There are two basic mechanisms that help the compiler do its job of translation. The compiler makes heavy use of a *symbol table* to keep track of variables during compilation. The compiler also follows a systematic partitioning of memory—it carefully allocates memory to these variables based on certain characteristics, with certain regions of memory reserved for objects of a particular class. In this section, we'll take a closer look at these two processes.

12.5.1 Symbol Table

In Chapter 7, we examined how the assembler systematically keeps track of labels within an assembly program by using a symbol table. Like the assembler, the C

```
#include <stdio.h>

int main()
{
  int amount;    /* The number of bytes to be transferred  */
  int rate;      /* The average network transfer rate      */
  int time;      /* The time, in seconds, for the transfer */

  int hours;     /* The number of hours for the transfer   */
  int minutes;   /* The number of mins for the transfer    */
  int seconds;   /* The number of secs for the transfer    */

  /* Get input: number of bytes and network transfer rate */
  printf("How many bytes of data to be transferred?  ");
  scanf("%d", &amount);

  printf("What is the transfer rate (in bytes/sec)?  ");
  scanf("%d", &rate);

  /* Calculate total time in seconds                      */
  time = amount / rate;

  /* Convert time into hours, minutes, seconds            */
  hours = time / 3600;          /* 3600 seconds in an hour */
  minutes = (time % 3600) / 60; /* 60 seconds in a minute */
  seconds = ((time % 3600) % 60); /* remainder is seconds */

  /* Output results */
  printf("Time : %dh %dm %ds\n", hours, minutes, seconds);
}
```

Figure 12.4 A C program that performs a simple network rate calculation

compiler keeps track of variables in a program with a symbol table. Whenever the compiler reads a variable declaration, it creates a new entry in its symbol table corresponding to the variable being declared. The entry contains enough information for the compiler to manage the storage allocation for the variable and generation of the proper sequence of machine code whenever the variable is used in the program. Each symbol table entry for a variable contains (1) its name, (2) its type, (3) the place in memory the variable has been allocated storage, and (4) an identifier to indicate the block in which the variable is declared (i.e., the scope of the variable).

Figure 12.5 shows the symbol table entries corresponding to the variables declared in the network rate calculation program in Figure 12.4. Since this program contains six variables declarations, the compiler ends up with six entries in its symbol table for them. Notice that the compiler records a variable's location in memory as an offset, with most offsets being negative. This offset indicates the relative position of the variable within the region of memory it is allocated.

Identifier	Type	Location (as an offset)	Scope	Other info...
amount	int	0	main	...
hours	int	–3	main	...
minutes	int	–4	main	...
rate	int	–1	main	...
seconds	int	–5	main	...
time	int	–2	main	...

Figure 12.5 The compiler's symbol table when it compiles the program from Chapter 11

12.5.2 Allocating Space for Variables

There are two regions of memory in which C variables are allocated storage: the *global data section* and the *run-time stack*.[2] The global data section is where all global variables are stored. More generally, it is where variables of the static storage class are allocated (we say more about this in Section 12.6). The run-time stack is where local variables (of the default automatic storage class) are allocated storage.

The offset field in the symbol table provides the precise information about where in memory variables are actually stored. The offset field simply indicates how many locations from the base of the section a variable is allocated storage.

For instance, if a global variable `earth` has an offset of 4 and the global data section starts at memory location 0x5000, then `earth` is stored in location 0x5004. All our examples of compiler-generated machine code use R4 to contain the address of the beginning of the global data section—R4 is referred to as the *global pointer*. Loading the variable `earth` into R3, for example, can be accomplished with the following LC-3 instruction:

```
LDR  R3, R4, #4
```

If `earth` is instead a local variable, say for example in the function `main`, the story is slightly more complicated. All local variables for a function are allocated in a "memory template" called an *activation record* or *stack frame*. For now, we'll examine the format of an activation record and leave the motivation for why we need it for Chapter 14 when we discuss functions. An activation record is a region of contiguous memory locations that contains all the local variables for a given function. Every function has an activation record (or more precisely, every invocation of a function has an activation record—more on this later).

[2] For examples in this textbook, all variables will be assigned a memory location. However, real compilers perform code optimizations that attempt to allocate variables in registers. Since registers take less time to access than memory, the program will run faster if frequently accessed values are put into registers.

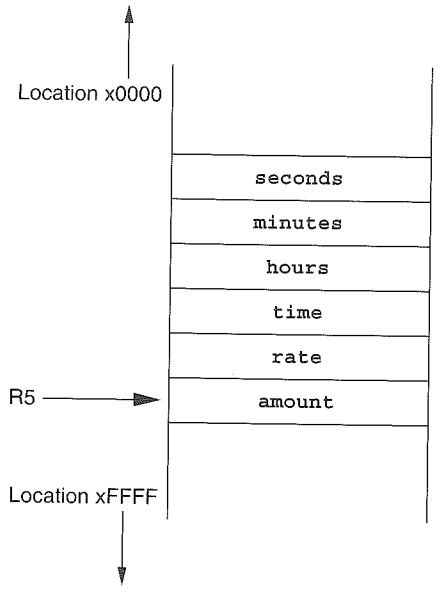


Figure 12.6 An example of an activation record in the LC-3's memory. This function has five local variables. R5 is the frame pointer and points to the first local variable

Whenever we are executing a particular function, the highest memory address of the activation record will be stored in R5—R5 is called the *frame pointer*. For example, the activation record for the function `main` from the code in Figure 12.4 is shown in Figure 12.6. Notice that the variables are allocated in the record in the reverse order in which they are declared. Since the variable `amount` is declared first, it appears nearest to the frame pointer R5.

If we make a reference to a particular local variable, the compiler will use the variable's symbol table entry to generate the proper code to access it. In particular, the offset in the variable's symbol table entry indicates where in the activation record the variable has been allocated storage. To access the variable `seconds`, the compiler would generate the instruction:

```
LDR  R0, R5, #-5
```

A preview of things to come: Whenever we call a function in C (in C, subroutines are called functions), the activation record for the function is pushed on to the run-time stack. That is, the function's activation record is allocated on top of the stack. R5 is appropriately adjusted to point to the base of the record—therefore any code within the function that accesses local variables will now work correctly. Whenever the function completes and control is about to return to the caller, the activation record is popped off the stack. R5 is adjusted to point to the caller's activation record. Throughout all of this, R6 always contains the address of the top of the run-time stack—it is called the *stack pointer*. We will revisit this in more detail in Chapter 14.

Figure 12.7 shows the organization of the LC-3's memory when a program is running. Many UNIX-based systems arrange their memory space similarly.

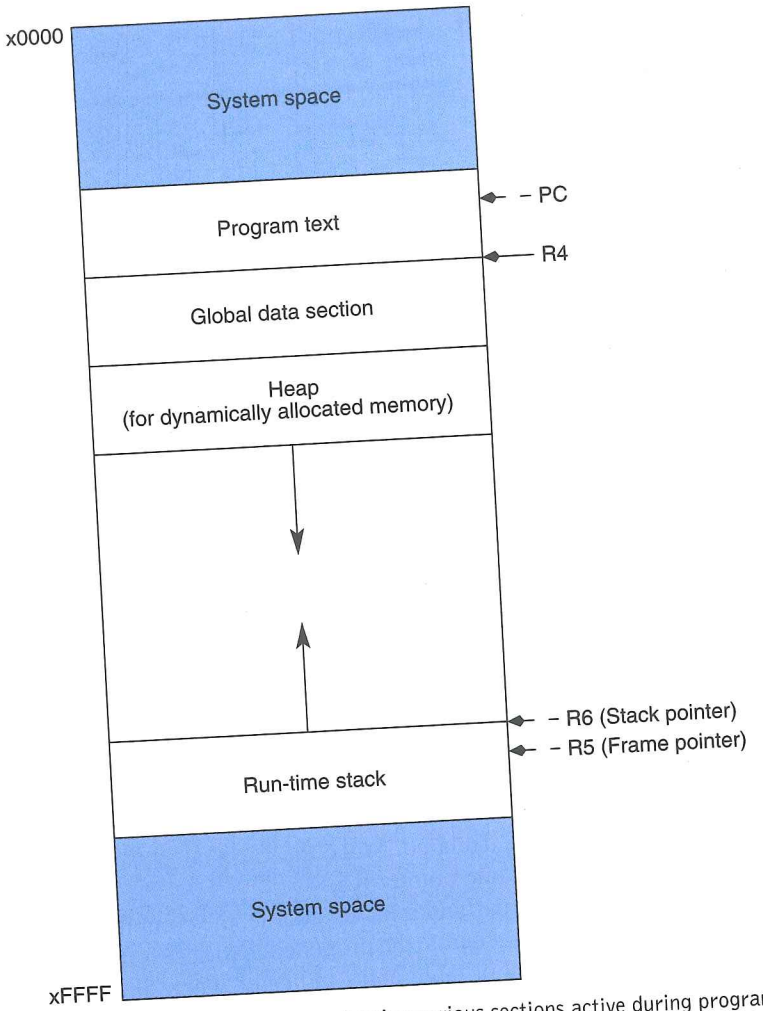


Figure 12.7 The LC-3 memory map showing various sections active during program execution

The program itself occupies a region of memory (labelled Program text in the diagram); so does the run-time stack and the global data section. There is another region reserved for dynamically allocated data called the *heap* (we will discuss this region in Chapter 19). Both the run-time stack and the heap can change size as the program executes. For example, whenever one function calls another, the run-time stack grows because we push another activation record onto the stack—in fact, it grows toward memory address x0000. In contrast, the heap grows toward 0xFFFF. Since the stack grows toward x0000, the organization of an activation record appears to be "upside-down": that is, the first local variable appears at the memory location pointed to by R5, the next one at R5 − 1, the subsequent one at R5 − 2, and so forth (as opposed to R5, R5 + 1, R5 + 2, etc).

During execution, the PC points to a location in the program text, R4 points to the beginning of the global data section, R5 points within the run-time stack,

and R6 points to the very top of the run-time stack. There are certain regions of memory, marked System space in Figure 12.7, that are reserved for the operating system, for things such as TRAP routines, vector tables, I/O registers, and boot code.

12.5.3 A Comprehensive Example

Now that we have examined the LC-3 compiler's techniques for tracking and allocating space for variables in memory, let's take a look at a comprehensive C example and its translation into LC-3 code.

Figure 12.8 is a C program that performs some simple operations on integer variables and then outputs the results of these operations. The program contains one global variable, `inGlobal`, and three local variables, `inLocal`, `outLocalA`, and `outLocalB`, which are local to the function `main`.

The program starts off by assigning initial values to `inLocal` and `inGlobal`. After the initialization step, the variables `outLocalA` and `outLocalB` are updated based on two calculations performed using `inLocal` and `inGlobal`. After the calculation step, the values of `outLocalA` and `outLocalB` are output using the `printf` library function. Notice because we are using `printf`, we must include the standard I/O library header file, `stdio.h`.

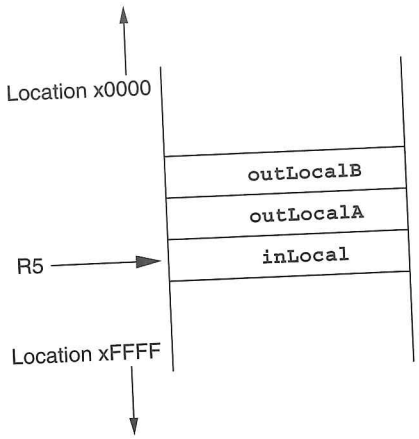
When analyzing this code, the LC-3 C compiler will assign the global variable `inGlobal` the first available spot in the global data section, which is at offset 0. When analyzing the function `main`, it will assign `inLocalA` to offset 0, `outLocalA` to offset −1, and `outLocalB` to offset −2 within `main`'s activation

```
/* Include the standard I/O header file */
#include <stdio.h>

int inGlobal;     /* inGlobal is a global variable because */
                  /* it is declared outside of all blocks  */

int main()
{
  int inLocal;    /* inLocal, outLocalA, outLocalB are all */
  int outLocalA; /* local to main                          */
  int outLocalB;

  /* Initialize */
  inLocal  = 5;
  inGlobal = 3;

  /* Perform calculations */
  outLocalA = inLocal & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* Print out results */
  printf("outLocalA = %d,outLocalB=%d\n",outLocalA,outLocalB);
}
```

Figure 12.8 A C program that performs simple operations

Identifier	Type	Location (as an offset)	Scope	Other info...
inGlobal	int	0	global	...
inLocal	int	0	main	...
outLocalA	int	–1	main	...
outLocalB	int	–2	main	...

(a) Symbol table



(b) Activation record for main

Figure 12.9 The LC-3 C compiler's symbol table when compiling the program in Figure 12.8 and the activation record format for its function `main`

record. A snapshot of the compiler's symbol table corresponding to this program along with the activation record of `main` are shown in Figure 12.9.

The resulting assembly code generated by the LC-3 C compiler is listed in Figure 12.10. Execution starts at the instruction labeled `main`.

12.6 Additional Topics

The last major section of this chapter involves a set of additional topics involving variables and operators. Some of the topics are advanced issues involving concepts we covered earlier in the chapter; some of the topics are miscellaneous features of C. We provide this section in order to complete our coverage of C, but this material is not essential to your understanding of the material in later chapters. For those of you interested in a more complete coverage of variables and operators in C, read on!

12.6.1 Variations of the Three Basic Types

C gives the programmer the ability to specify larger or smaller sizes for the three basic types `int`, `char`, and `double`. The modifiers `long` and `short` can be attached to `int` with the intent of extending or shortening the default size. For example, a `long int` can declare an integer that has twice the number of bits as a regular `int`, thereby allowing us to represent a larger range of integers in a C program. Similarly, the specifier `long` can be attached to the `double` type to create a larger floating point type (if supported by the particular system) with greater range and precision.

```
1   main:
2     :
3     :
4     <startup code>
5     :
6     :
7     AND  R0, R0, #0
8     ADD  R0, R0, #5   ;  inLocal is at offset 0
9     STR  R0, R5, #0   ;  inLocal = 5;
10
11    AND  R0, R0, #0
12    ADD  R0, R0, #3   ;  inGlobal is at offset 0, in globals
13    STR  R0, R4, #0   ;  inGlobal = 3;
14
15    LDR  R0, R5, #0   ;  get value of inLocal
16    LDR  R1, R4, #0   ;  get value of inGlobal
17    NOT  R1, R1       ;  ~inGlobal
18    AND  R2, R0, R1   ;  calculate inLocal & ~inGlobal
19    STR  R2, R5, #-1  ;  outLocalA = inLocal & ~inGlobal;
20                      ;  outLocalA is at offset -1
21
22    LDR  R0, R5, #0   ;  get value of inLocal
23    LDR  R1, R4, #0   ;  get value of inGlobal
24    ADD  R0, R0, R1   ;  calculate inLocal + inGlobal
25
26    LDR  R2, R5, #0   ;  get value of inLocal
27    LDR  R3, R4, #0   ;  get value of inGlobal
28    NOT  R3
29    ADD  R3, R3, #1   ;  calculate -inGlobal
30
31    ADD  R2, R2, R3   ;  calculate inLocal - inGlobal
32    NOT  R2
33    ADD  R2, R2, #1   ;  calculate -(inLocal - inGlobal)
34
35    ADD  R0, R0, R2   ;  (inLocal + inGlobal) - (inLocal - inGlobal)
36    STR  R0, R5, #-2  ;  outLocalB = ...
37                      ;  outLocalB is at offset -2
38    :
39    :
40    <code for calling the function printf>
41    :
42    :
```

Figure 12.10 The LC-3 code for the C program in Figure 12.8

The modifier `short` can be used to create variables that are smaller than the default size, which can be useful when trying to conserve on memory space when handling data that does not require the full range of the default data type. The following example demonstates how the variations are declared:

```
long double particlesInUniverse;
long int worldPopulation;
short int ageOfStudent;
```

Because the size of the three basic C types is closely tied to the types supported by the underlying ISA, many compilers only support these modifiers `long` and `short` if the computer's ISA supports these size variations. Even though a variable can be declared as a `long int`, it may be equivalent to a regular `int` if the underlying ISA has no support for longer versions of the integer data type. See Appendix D.3.2 for more examples and additional information on `long` and `short`.

Another useful variation of the basic `int` data type is the unsigned integer. We can declare an unsigned integer using the `unsigned` type modifier. With unsigned integers, all bits are used to represent nonnegative integers (i.e., positive numbers and zero). In the LC-3 for instance, which has 16-bit integers, an unsigned integer has a value between 0 and 65,535. When dealing with real-world objects that by nature do not take on negative values, unsigned integers might be the data type of choice. The following are examples of unsigned integers:

```
unsigned int numberOfDays;
unsigned int populationSize;
```

Following are some sample variations of the three basic types:

```
long int ounces;
short int gallons;
long double veryVeryLargeNumber = 4.12936E361;
unsigned int sizeOfClass = 900;
float oType = 9.24;
float tonsOfGrain = 2.998E8;
```

12.6.2 Literals, Constants, and Symbolic Values

In C, variables can also be declared as *constants* by adding the `const` qualifier before the type specifier. These constants are really variables whose values do not change during the execution of a program. For example, in writing a program that calculates the area and circumference of a circle of a given radius, it might be useful to create a floating point constant called `pi` initialized to the value 3.14159. Figure 12.11 contains an example of such a program.

This example is useful for making a distinction between three types of constant values that often appear in C code. *Literal* constants are unnamed values that appear *literally* in the source code. In the circle example, the values `2` and `3.14159` are examples of *literal* constants. In C, we can represent literal constants in hexadecimal by prepending a `0x` in front of them, for example `0x1DB`. ASCII literals require single quotes around them, as for example `'R'`, which is the ASCII value of the character R. Floating point literals can be the exponential notation described in Section 12.2.1. An example of the second type of constant value is `pi`, which is declared as a constant value using a variable declaration with the `const` qualifier. The third type of constant value is created using the preprocessor directive `#define`, an example of which is the symbolic value `RADIUS`. All three types create values that do not change during the execution of a program.

```
1  #include <stdio.h>
2
3  #define RADIUS  15.0      /* This value is in centimeters */
4
5  int main()
6  {
7    const double pi = 3.14159;
8    double area;
9    double circumference;
10
11   /* Calculations */
12   area = pi * RADIUS * RADIUS;          /* area = pi*r^2 */
13
14   circumference = 2 * pi * RADIUS;     /* circumference = */
15                                        /*          2*pi*r */
16
17   printf("Area of a circle with radius %f cm is %f cm^2\n",
18          RADIUS, area);
19
20   printf("Circumference of the circle is %f cm\n",
21         circumference);
22 }
```

Figure 12.11 A C program that computes the area and circumference of a circle with a radius of 15 cm

The distinction between constants declared using `const` and symbolic values defined using `#define` might seem a little subtle to you. Using one versus another is really a matter of programming style rather than function. Declared constants are used for things we traditionally think of as constant values, which are values that never change. The constant `pi` is an example. Physical constants such as the speed of light, or the number of days in a week, are conventionally represented by declared constants.

Values that stay constant during a single execution of the program but which might be different from user to user, or possibly from invocation to invocation, are represented by symbolic values using `#define`. Such values can be thought of as parameters for the program. For example, `RADIUS` in Figure 12.11 can be changed and the program recompiled, then re-executed.

In general, naming a constant using `const` or `#define` is preferred over leaving the constant as a literal in your code. Names convey more meaning about your code than unnamed literal values.

12.6.3 Storage Class

Earlier in the chapter, we mentioned three basic properties of a C variable: its identifier, its type, and its scope. There is another: *storage class*. The storage class of a variable indicates how the C compiler allocates its storage, and in particular indicates whether or not the variable loses its value when the block that contains it has completed execution. There are two storage classes in C: *static* and *automatic*.

Static variables retain their values between invocations. Automatic variables lose their values when their block terminates. In C, global variables are of static storage class, that is, they retain their value until the program ends. Local variables are by default of automatic storage class. Local variables can be declared as static class variables by using the `static` modifier on the declaration. For example, the variable declared by `static int localVar;` will retain its value even when its function completes execution. If the function is executed again (during the same program execution), `localVar` will retain its previous value. In particular, the use of the `static` keyword on a local variable causes the compiler to allocate storage for the variable in the global data section, while keeping it private to its block. See Appendix D.3.3 for additional examples on storage class.

12.6.4 Additional C Operators

The C programming language has a collection of unusual operators, which have become a trademark of C programming. Most of these operators are combinations of operators we have already seen. The combinations are such that they make expressing commonly used computations even simpler. However, to someone who is not accustomed to the shorthand notation of these operators, reading and trying to understand C code that contains them can be difficult.

Assignment Operators

C also allows certain arithmetic and bitwise operators to be combined with the assignment operator. For instance, if we wanted to add 29 to variable `x`, we could use the shorthand operator `+=` as follows:

```
x  += 29;
```

This code is equivalent to

```
x  = x + 29;
```

Table 12.6 lists some of the special operators provided by C. The postfix operators have highest precedence, followed by prefix. The assignment operators have lowest precedence. Each group associates from right to left.

Table 12.6 Assignment Operators in C

Operator symbol	Operation	Example usage
`+=`	add and assign	`x += y`
`-=`	subtract and assign	`x -= y`
`*=`	multiply and assign	`x *= y`
`/=`	divide and assign	`x /= y`
`%=`	modulus and assign	`x %= y`
`&=`	and and assign	`x &= y`
`|=`	or and assign	`x |= y`
`^=`	xor and assign	`x ^= y`
`«=`	left-shift and assign	`x «= y`
`»=`	right-shift and assign	`x »= y`

More examples are as follows:

```
h += g;          /* Equivalent to h = h + g;   */
h %= f;          /* Equivalent to h = h % f;   */
h <<= 3;         /* Equivalent to h = h << 3;  */
```

Conditional Expressions

Conditional expressions are a unique feature of C that allow for simple decisions to be made with a simple expression. The symbols for the conditional expression are the question mark and colon, `?` and `:`. The following is an example:

```
x = a ? b : c;
```

Here variable `x` will get either the value of `b` **or** the value of `c` based on the logical value of `a`. If `a` is nonzero, `x` will get the value of `b`. Otherwise, it will get the value of `c`.

Figure 12.12 is a complete program that uses a conditional expression to calculate the maximum of two integers. The maximum of these two input values is determined by a conditional expression and is assigned to the variable `maxValue`. The value of `maxValue` is output using `printf`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5     int maxValue;
6     int input1;
7     int input2;
8
9     printf("Input an integer: ");
10    scanf("%d", &input1);
11    printf("Input another integer: ");
12    scanf("%d", &input2);
13
14    maxValue = (input1 > input2) ? input1 : input2;
15    printf("The larger number is %d\n", maxValue);
16 }
```

Figure 12.12 A C program that uses a conditional expression

12.7 Summary

We conclude this chapter by summarizing the three key concepts we covered.

- **Variables in C.** The C programming language supports variables of three basic types: integers (`int`), characters (`char`), and floating point numbers (`double`). C, like all other high-level languages, provides the programmer the ability to provide symbolic names to these variables. Variables in C can be locally

declared within a block of code (such as a function) or globally visible by all blocks.

- **Operators in C.** C's operators can be categorized by the function they perform: assignment, arithmetic, bitwise manipulations, logical and relational tests. We can form expressions using variables and operators such that the expressions get evaluated according to precedence and associativity rules. Expressions are grouped into statements, which express the work the program is to perform.
- **Translating C Variables and Operators into LC-3 Code.** Using a symbol table to keep track of variable declarations, a compiler will allocate local variables for a function within an activation record for the function. The activation record for the function is pushed onto the run-time stack whenever the function is executed. Global variables in a program are allocated in the global data section.

Exercises

12.1 Generate the compiler's symbol table for the following code. Assume all variables occupy one location in memory.

```
{
   double ff;
   char cc;
   int ii;
   char dd;
}
```

12.2 The following variable declaration appears in a program:

```
int r;
```

a. If `r` is a local variable, to what value will it be initialized?
b. If `r` if a global variable, to what value will it be initialized?

12.3 What are the ranges for the following two variables if they are stored as 32-bit quantities?

```
int plusOrMinus;
unsigned int positive;
```

12.4 Evaluate the following floating point literals. Write their values in standard decimal notation.

a. 111 E −11
b. −0.00021 E 4
c. 101.101 E 0

12.5 Write the LC-3 code that would result if the following local variable declarations were compiled using the LC-3 C compiler:

```
char c = 'a';
int  x = 3;
int  y;
int  z = 10;
```

12.6 For the following code, state the values that are printed out by each `printf` statement. The statements are executed in the order A, B, C, D.

```
int t;  /* This variable is global */

{
   int t = 2;

   printf("%d\n", t);      /*  A  */
   {
      printf("%d\n", t);   /*  B  */
      t = 3;
   }
   printf("%d\n", t);      /*  C  */
}

{
   printf("%d\n", t);      /*  D  */
}
```

12.7 Given that `a` and `b` are both integers where `a` and `b` have been assigned the values 6 and 9, respectively, what is the value of each of the following expressions? Also, if the value of `a` or `b` changes, give their new value.

a. `a | b`
b. `a || b`
c. `a & b`
d. `a && b`
e. `!(a + b)`
f. `a % b`
g. `b / a`
h. `a = b`
i. `a = b = 5`
j. `++a + b--`
k. `a = (++b < 3) ? a : b`
l. `a <<= b`

12.8 For the following questions, write a C expression to perform the following relational test on the character variable `letter`.

a. Test if `letter` is any alphabetic character or a number.
b. Test if `letter` is any character except an alphabetic character or a number.

12.9 *a.* What does the following statement accomplish? The variable `letter` is a character variable.

```
letter = ((letter >= 'a' && letter <= 'z') ? '!' : letter);
```

b. Modify the statement in (a) so that it converts lowercase to uppercase.

12.10 Write a program that reads an integer from the keyboard and displays a 1 if it is divisible by 3 or a 0 otherwise.

12.11 Explain the differences between the following C statements:

a. `j = i++;`
b. `j = ++i;`
c. `j = i + 1;`
d. `i += 1;`
e. `j = i += 1;`
f. Which statements modify the value of `i`? Which ones modify the value of `j`? If `i = 1` and `j = 0` initially, what will the values of `i` and `j` be after each statement is run separately?

12.12 Say variables `a` and `b` are both declared locally as `long int`.

a. Translate the expression `a + b` into LC-3 code, assuming a `long int` occupies two bytes. Assume `a` is allocated at offset 0 and `b` is at offset −1 in the activation record for their function.

b. Translate the same expression, assuming a `long int` occupies four bytes, `a` is allocated offset 0, and `b` is at offset −2.

12.13 If initially, `a = 1, b = 1, c = 3`, and `result = 999`, what are the values of the variables after the following C statement is executed?

```
result = b + 1 | c + a;
```

12.14 Recall the machine busy example from Chapter 2. Say the integer variable `machineBusy` tracks the busyness of all 16 machines. Recall that a 0 in a particular bit position indicates the machine is busy and a 1 in that position indicates the machine is idle.

a. Write a C statement to make machine 5 busy.
b. Write a C statement to make machine 10 idle.
c. Write a C statement to make machine `n` busy. That is, the machine that has become busy is an integer variable `n`.
d. Write a C expression to check if machine 3 is idle. If it is idle, the expression returns a 1. If it is busy, the expression returns a 0.
e. Write a C expression that evaluates to the number of idle machines. For example, if the binary pattern in `machineBusy` were `1011 0010 1110 1001`, then the expression will evaluate to 9.

12.15 What purpose does the semicolon serve in C?

12.16 Say we are designing a new computer programming language that includes the operators `@`, `#`, `$` and `U`. How would the expression `w @ x # y $ z U a` get evaluated under the following constraints?

a. The precedence of `@` is higher than `#` is higher than `$` is higher than `U`. Use parentheses to indicate the order.
b. The precedence of `#` is higher than `U` is higher than `@` is higher than `$`.
c. Their precedence is all the same, but they associate left to right.
d. Their precedence is all the same, but they associate right to left.

12.17 Notice that the C assignment operators have the lowest precedence. Say we have developed a new programming language called *Q* that works exactly like C, except that the assignment operator had the highest precedence.

a. What is the result of the following Q statement? In other words, what would the value of `x` be after it executed?

```
x = x + 1;
```

b. How would we change this Q statement so that it works the same way as it would in C?

12.18 Modify the example program in Chapter 11 (Figure 11.3) so that it prompts the user to type a character and then prints every character from that character down to the character `!` in the order they appear in the ASCII table.

12.19 Write a C program to calculate the sales tax on a sales transaction. Prompt the user to enter the amount of the purchase and the tax rate. Output the amount of sales tax and the total amount (including tax) on the whole purchase.

12.20 Suppose a program contains the two integer variables `x` and `y`, which have values 3 and 4, respectively. Write C statements that will exchange the values in `x` and `y` so that after the statements are executed, `x` is equal to 4 and `y` is equal to 3.

a. First, write this routine using a temporary variable for storage.
b. Now rewrite this routine without using a temporary variable.